A Web Service Composition Method Based on Compact K^2-trees

Jing Li[a], Yuhong Yan[b]
Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Canada
Email: jing.li.hnu@gmail.com[a]
Email: yuhong@encs.concordia.ca[b]

Daniel Lemire
LICEF Research Center, TELUQ
Université du Québec
Montreal, Canada
Email: lemire@gmail.com

***Abstract*—With the advent of cloud computing, a significant number of web services are available on the Internet. Services can be combined together when user's requirements are too complex to be solved by individual services. Since there are many services, searching a solution may require much storage. We propose to apply a compact data structure to represent the web service composition graph. To the best of our knowledge, our work is the first attempt to consider compact structure in solving the web service composition problem. Experimental results show that our method can find a valid solution to the composition problem; meanwhile, it takes less space and shows good scalability when handling a large number of web services.**

***Index Terms*—QoS-aware service composition; graph compression; semantic match;**

I. Introduction

Cloud computing allows users to benefit from existing resources on the Internet without investing in new infrastructure. Service-Oriented Computing (SOC) enables composition of existing services on the Internet to achieve complex functionality. There has been considerable research on the service composition problem. Among these, planning has been successfully applied to solve composition problem [1], [2], [3]. Planning selects suitable actions and orders them in sequence to achieve the goal [4]. To solve a service composition problem, a planning algorithm first constructs a search graph from the initial state to the goal state, it then finds a solution path by applying a backward search. Since the number of services and combination possibilities is huge, planning algorithms are limited by the search space and may fail to find a solution.

To deal with this challenge, we use a compressed graph representation. Memory-efficient graph representations have been widely studied [5], [6]. In particular, Brisaboa *et al.,* introduce a compact tree to represent the adjacency matrix of the web graph (the K^2-tree) [7]. Experimental results show their method offers a good compression rate. We apply this data structure to the web service composition problem. This allows us to handle the service composition problem with a smaller storage requirement. Experimental results verify the feasibility and effectiveness of our method.

The organization of this paper is as follows: Section II shows how to represent a web service composition graph with a compact K-ary tree and how to traverse the tree. Search algorithms are given in Section III. We present our experimental results in Section IV. Section V reviews related work and the conclusion is drawn in Section VI.

II. Preliminary

A. QoS-aware service composition

A web service w is defined as a tuple with the following components:

- w_{in} is a finite set of typed input parameters of w. A web service is invoked only when all its input parameters are satisfied.
- w_{out} is a finite set of typed output parameters of w. We refer to the input and output types as *concepts*. OWL-S (Web Ontology Language for Web Services [8]) files are used to define relationships between services and concepts.
- Q is a finite set of quality-of-service (QoS) valies w. The criteria for QoS are determined from users' constraints and preferences. Following [9] and [2], we use response time and throughput to measure the QoS value of a service. Response time in a data system is the interval between the arrival of the request and the beginning of delivery the response (unit: milliseconds). Throughput is the average rate of successful message delivered per time unit over a communication channel (unit: requests/min).

A web service composition problem can be represented by a tuple (S, C_{in}, C_{out}, Q) with the following components:

- S is a finite set of services.
- C_{in} is a finite set of typed input parameters.
- C_{out} is a finite set of typed output parameters.
- Q is a finite set of quality criteria.

In semantic service composition, we use plug-in matching degree to match services. An ontology rooted tree is built to represent relationships of concepts (outputs of services). We extend the output concepts of services with ancestors in the ontology rooted tree. Two services can be connected if the input of a service is a subset of the output of the other service. This semantic model, borrowed from [9], is minimalist but we leave richer models to future work. Also, this model is consistent

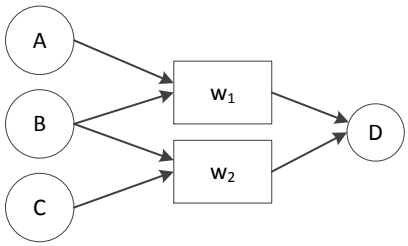

Fig. 1. A web service composition representation

TABLE I
ADJACENCY MATRIX OF FIGURE 1

	w_1	w_2
A	1	0
B	1	1
C	0	1
D	2	2

with many service composition approaches, e.g., [10], [11], [12].

Suppose services are represented by $w_1, w_2, ..., w_n$, services can be connected in sequence or parallel. Services in the sequence control structure are invoked one by one $(w_1; w_2; ...; w_n)$. Services in flow control are invoked in parallel $(w_1||w_2||...||w_n)$. When the service composition must satisfy both the functional requirements and the QoS constraints, we say that we have a QoS-aware service composition problem.

B. Web service composition representation

A web service composition problem can be represented by a directed graph where services and parameters are seen as nodes and their relationships are represented as directed edges. Figure 1 is a simple example of a web service composition graph. We use circles to represent parameters and rectangles to represent services. If a parameter p is an input or output of a service w, there is an edge between p and w, we use arrows to represent edges. For example, in this figure, A is an input parameter of w and w has an output parameter D. No cycle or parallel edges exist in a web service composition graph.

We use an adjacency matrix to represent the relationships between services and parameters. Each row represents a parameter and each column represents a service. The cell $\mathrm{adjMatrix}[i][j]$ represents the edge between node i and j. We have that $\mathrm{adjMatrix}[i][j]$ = 1 if parameter i is an input parameter of service j (j is an output service of parameter i), $\mathrm{adjMatrix}[i][j]$ = 2 if parameter i is an output parameter of service j (j is an input service of parameter i), otherwise, $\mathrm{adjMatrix}[i][j]$ = 0. Figure 1 can be represented by the adjacency matrix shown in Table I. For example, the edge from A to w_1 is represented as $\mathrm{adjMatrix}[1][1] = 1$, the edge from service w_1 to D is represented as $\mathrm{adjMatrix}[4][1] = 2$.

TABLE II
EXTENDED ADJACENCY MATRIX OF FIGURE 1

	w_1	w_2	X	X
A	1	0	0	0
B	1	1	0	0
C	0	1	0	0
D	2	2	0	0

C. Compact graph representation

There can be thousands of services, but each service is only expected to have, at most, dozens of input and output concepts. Thus the adjacency is necessarily sparse. We want to use a data structure to efficiently represent such a sparse matrix. For this purpose, let us briefly present the key ideas from Brisaboa *et al.* [13]. They rely on the notion of K-ary tree.

Definition 1. *A K-ary tree is a rooted tree in which each node has no more than k children.*

Definition 2. *A full K-ary tree is a K-ary tree where in each level every node has either 0 or k children.*

We use two compact K^2-trees to represent the adjacency matrix of services and parameters. One for the edges from input parameters to services (inTree), the other for the edges from services to their output parameters ($\mathrm{outTree}$). The code idea of K^2-tree compression is as follows. Assume temporarily that the matrix we want to compress is square and that its size is a power of k. The whole matrix is represented as the root of tree with value 1. Then, the matrix is divided from left to right into k^2 distinct sub-matrices, each with size n^2/k^2. Each of the sub-matrix is a child of the root node, so the root has exactly k^2 children. The value of the child node is a leaf node with value 0 if it is filled with zeroes, otherwise, the node is an internal node with value 1. For each internal node, we recursively divide the sub-matrix it represents into k^2 parts, this process ends when the size of the sub-matrix is 1 or the sub-matrices are all 0. Because sub-matrices filled with zeroes can be stored as a single value (0), the K^2-tree enables good compression. If n is not a power of k, the adjacency matrix is extended to right and bottom with 0, so that the extended matrix is a square matrix and the length n' is a power of k. The size of the extended adjacency matrix is $n' \times n'$. This will not bring significant changes because large area of 0 can be seen as a sub-matrix and represented by a leaf node in the tree. If we take $k = 2$, the adjacency matrix in Table I is extended to a 4×4 matrix as shown in Table II. Figure 2 and Figure 3 are two compact trees to represent the adjacency matrix corresponding to inTree and outTree.

D. Navigating in a K^2-tree

To find the output services of a given parameter p, we need to find the 1s in row p of inTree and their corresponding column numbers. This is done by a level order traversal of inTree. Similarly, to obtain the input services of a given parameter p, we need to find the 2s in row p of $\mathrm{outTree}$.

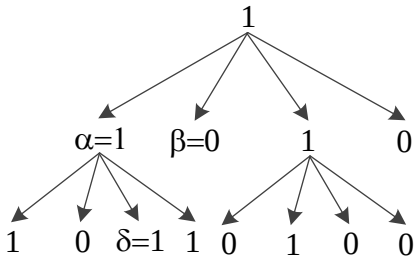

Fig. 2. Compact K^2-tree representation from input parameters to services (inTree)

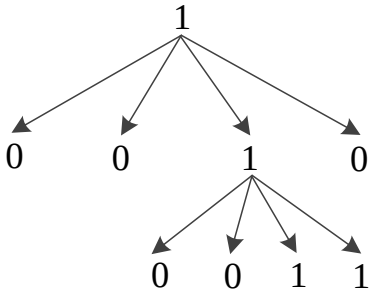

Fig. 3. Compact K^2-tree representation from services to output parameters (outTree)

Example 1. *To find the output services of parameter A in Table II, we start from the root node in figure 2. As $k = 2$, from the way the K^2-tree is constructed, we know that first k children nodes (α and β) of root are related to row 1, α has children because its value is 1. The first k children of α represent cells in row 1, their values are 1 and 0 and β does not have any child because its value is 0. Thus, A has an output w_1. Each internal node has k^2 children, so we need to check k children nodes in each iteration.*

III. Data structure and algorithm

Algorithm 1 ***TreeBuild*** builds the K^2-tree to represent the adjacency matrix $matrix$ whose length is n. We first create a root node of the tree with value 1 and add the root into the linked list pos. The linked list pos stores all the nodes with value 1, then, we extend the tree level by level by adding the k^2 children of each node in pos. This algorithm stops when the size of sub-matrix is 1.

Theorem 1. *To represent the adjacency matrix, the space required for the compact tree is: $k^2(\lceil \log_k n \rceil (e_{\text{in}} + e_{\text{out}}) - (e_{\text{in}} \log_{k^2} e_{\text{in}} + e_{\text{out}} \log_{k^2} e_{\text{out}}))$.*

Proof: In the web service composition problem, the number of concepts is always bigger than the number of services, so the size of the adjacent matrix is decided by the number of concepts. Suppose we have n concepts, before compacting, the storage requirement is n^2 for the adjacency matrix. Suppose we have e_{in} edges from input concept to services, and e_{out} output concepts from services. We review the worst possible inTree. The height of the tree is $h = \lceil \log_k n \rceil$, if $e_{\text{in}} = 1$, the space required is $k^2 \lceil \log_k n \rceil$; if $e_{\text{in}} = 2$, and in the worst case,

Algorithm 1 *TreeBuild*

Input: $matrix$: the adjacency matrix;
Output: $tree$: the compact tree;

```
1: create a root with value 1 for the tree;
2: pos ← root;
3: tempos ← φ
4: for length ← n/k; length ≥ 1; length ← length/k do
5:   while pos ≠ φ do
6:     parent ← pos.remove(pos.first)
7:     if parent = 1 then
8:       startRow ← calRow(tree, parent, n)
9:       startColumn ← calColumn(tree, parent, n)
10:      create k^2 children nodes child for parent, the sub-
         matrix has size length^2, the offset of the sub-
         matrix relative to the matrix is represented by
         startRow, startColumn and length
11:      if child = 1 then
12:        tempos ← tempos ∪ child
13:      end if
14:    end if
15:  end while
16:  while tempos ≠ φ do
17:    pos ← pos ∪ tempos.remove(tempos.first)
18:  end while
19: end for
20: return tree
```

a new internal node and its corresponding children are created, the space required is $k^2 \lceil \log_k n \rceil + k^2(\lceil \log_k n \rceil - 1)$ thus, at level l, the total space required is $k^2 e_{\text{in}}(\lceil \log_k n \rceil - l)$. There are at most $(k^2)^l$ nodes, and the internal nodes of the first $l-1$ levels are 1, thus $l = \log_{k^2} m$. The total space required for the tree is $k^2 e_{\text{in}}(\lceil \log_k n \rceil - \log_{k^2} e_{\text{in}})$.

Similarly, the total space for outTree is $k^2 e_{\text{out}}(\lceil \log_k n \rceil - \log_{k^2} e_{\text{out}})$. Thus, the total space required is $k^2(\lceil \log_k n \rceil (e_{\text{in}} + e_{\text{out}}) - (e_{\text{in}} \log_{k^2} e_{\text{in}} + e_{\text{out}} \log_{k^2} e_{\text{out}})$. ■

Algorithm 2 ***CalRow*** and Algorithm 3 ***CalColumn*** show how to calculate the position of the upper-left cell of each sub-matrix. The length of matrix is denoted as n, $tNode$ is the current node. If the current length of matrix is l, and the node is the first k children of its parent, the row offset of the upper-left cell of the sub-matrix to the parent matrix is 0, otherwise, the row offset is calculated as

$$offset = l/k. \tag{1}$$

In the tree, the root node maps to the whole $matrix$, the tree node in depth d maps to the sub-matrix of length

$$l = matrix.length/k^{d-1}. \tag{2}$$

Combining Equation 1 and Equation 2, we get $offset = matrix.length/k^d$.

Example 2. *We want to find the row of δ (its depth is 2) in Figure 2. This node is not the first or the second child of its parent, so $startRow = 1$ (Algorithm 2 line 8). In the second*

Algorithm 2 *CalRow*

Input: $tree, tNode, n$;
Output: $startRow$: the offset of sub-matrix;

```
1: startRow ← 0
2: temp ← tNode
3: while temp ≠ root do
4:    if temp is the first k children of parent then
5:       startRow ← startRow
6:    else
7:       set ← tree.depth(temp)
8:       startRow ← startRow + n/k^set
9:    end if
10:   temp ← tree.parent(temp)
11: end while
12: return startRow
```

round, α is δ's parent, as it is the first child of root, $startRow$ keeps the same. In the third round, $temp = root$, the algorithm stops. The row of δ is 1.

Algorithm 3 *CalColumn*

Input: $tree, tNode, n$
Output: $startColumn$: the offset of sub-matrix;

```
1: startColumn ← 0
2: temp ← tNode
3: while temp ≠ root do
4:    if temp is an odd child of parent then
5:       startColumn ← startColumn
6:    else
7:       set ← tree.depth(temp)
8:       startColumn ← startColumn + n/k^set
9:    end if
10:   temp ← tree.parent(temp)
11: end while
12: return startCol
```

Example 3. *We want to find the column of δ in Figure 2. This node is the third child of its parent, so $startColumn = 0$ (Algorithm 3 line 5). In the second round, α is δ's parent, as it is the first child of root, $startColumn$ keeps the same. In the third round, $temp = root$, the algorithm stops. The column of δ is 0.*

Algorithm 4 ***Direct*** describes how to retrieve direct nodes. To find the output service of parameter p, we need to locate the 1 in row p of the extended adjacency matrix. We use two arrays $ParentList$ (represented as P) and $LeafList$ (represented as L) to represent the compressed tree by a level order traversal. $ParentList$ stores values of nodes except the last level. $LeafList$ stores values of nodes in the last level. For example, to represent the tree of Figure 2, $ParentList$ = 1010, $LeafList$ = 10110100. $rank(P, i)$ count how many 1s appear in $ParentList$ until position i, $rank(P, -1) = 0$. If $P[x] = 1$, the children of x is at position $rank(P, x) \times k^2$ to $rank(P, x) \times k^2 + (k^2 - 1)$ [7]. $size$ denotes the length of current sub-matrix, row and col represent the row and column of matrix. $index$ represents the position in $P : L$.

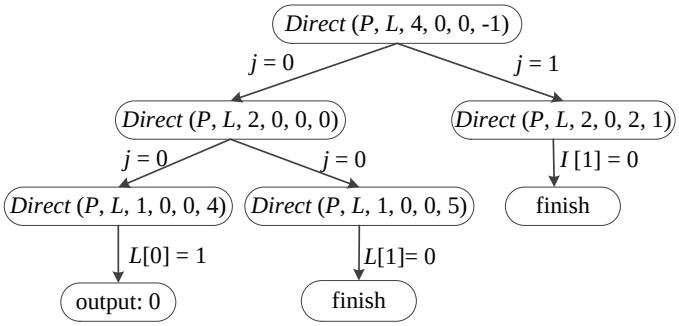

Fig. 4. Recursion trace of finding output service of A in Figure 1

If $index$ moves to array L and values 1, this means that the value of $cell[row][col]$ is not 0, and we add the current column col into the solution (lines 2–4). If the size of sub-matrix is not 0, and this is the first round ($index = -1$) or the node value is 1, which means the sub-matrix is not all 0, and check the children nodes of the current node (lines 10–11).

Algorithm 4 *Direct*

Input: $P, L, size, row, col, index$
Output: $colSol$:a set of services;

```
1: temp ← 0
2: if index ≥ P.length then
3:    if L[index − P.length] = 1 then
4:       colSol ← colSol ∪ col
5:    end if
6: else if size ≥ 1 then
7:    if index = −1||P[index] = 1 then
8:       temp ← k^2 × rank(P, index) + k × ⌊k × row/size⌋
9:    end if
10:   for j ← 0; j < k; j++ do
11:      direct(P, L, size/k, row%(size/k), col + j × (size/k), temp + j)
12:   end for
13: end if
14: return colSol
```

Example 4. *To find the output service of A, we refer to the cells containing 1s in row 0 of the matrix. We recursively search* inTree *with the beginning of* $Direct$(P, L,*4,0,0,-1), here* P *(resp.* L*) refers to the* ParentList *(resp.*LeafList*) of* inTree*. The searching process is shown in Figure 4. The output refers to service w_1. Similarly, to find the input service of parameter p', we need to search the* outTree *and locate the 2s in row p'.*

Algorithm 5 ***Reverse*** describes how to retrieve reverse nodes. If $index$ moves to array $LeafList$ and values 1, add the current row row into the solution (lines 2–4). If the size of sub-matrix is not 0, and this is the first round or the node

value is 1 (line 7), check the children nodes of the current node (lines 10–11).

Algorithm 5 *Reverse*

Input: $P, L, size, col, row, index$
Output: $rowSol$:a set of parameters;
1: $temp \leftarrow 0$
2: **if** $index \geq P.length$ **then**
3: **if** $L[index - P.length] = 1$ **then**
4: $rowSol \leftarrow rowSol \cup row$
5: **end if**
6: **else if** $size \geq 1$ **then**
7: **if** $index = -1 || P[index] = 1$ **then**
8: $temp \leftarrow k^2 \times rank(P, index) + \lfloor k \times col/size \rfloor$
9: **end if**
10: **for** $j \leftarrow 0; j < k; j++$ **do**
11: $reverse(P, L, size/k, col\%(size/k), row + j \times (size/k), temp + k \times j)$
12: **end for**
13: **end if**
14: **return** $rowSol$

Example 5. *To find the input parameters of w_2, we refer to the cells containing 1s in column 1 of the matrix. We recursively search the* inTree *with the beginning of* $Reverse(\mathrm{P}, \mathrm{L}$*,4,1,0,-1), the searching process is shown in Figure 5, the output refers to parameters B and C. Similarly, to find the output parameter of service w', we need to locate the 2s in column w' of the extended adjacency matrix and search the* outTree.

Algorithm 6 ***ForwardPropagation*** finds a set of evocable services in the web service composition problem. $ConceptPool$ stores all the concepts known so far. In each iteration, for each newly added concept, we find the services which use these added concepts as input, then for each service, if all its input concepts exist in the $conceptPool$, this service can be invoked and added in the $addedService$ set. We use $inTreeP$ (resp. $inTreeL$) to represent $ParentList$ (resp. $LeafList$) of $inTree$, similarly, $outTreeP$ (resp. $outTreeL$) represents $ParentList$ (resp. $LeafList$) of $outTree$. The algorithm ends when there are no more concepts to be added into the $conceptPool$. If all the output concepts of the service exist in the $conceptPool$, the service is removed from the $addedService$ set. Then we add the output concepts of all the services in $addedService$ into the $conceptPool$. To carry out a global optimization, we record the QoS values of services and corresponding concepts in the process of forward expand. For each service s, the total QoS value along the way will be calculated and stored as $s.curQoS$, for each concept c, the optimal QoS value is stored as $c.optQoS$, the service which provide the best QoS value to concept c is recorded as $c.optSrv$. If the goals are contained in $conceptPool$, we say there is a solution to the problem, we use a backtracking method to extract the solution, if not, there is no solution to the problem.

Algorithm 6 *ForwardPropagation*

Input: $initial, goal$
Output: Boolean:
1: $conceptPool \leftarrow initial$
2: $addedConcept \leftarrow initial$
3: **while** $addedConcept \neq \varphi$ **do**
4: **for** each concept c in $addedConcept$ **do**
5: $tempSrv \leftarrow direct(\mathrm{inTreeP}, \mathrm{inTreeL}, n, c, 0, -1)$
6: $addedService \leftarrow adddedService \cup tempSrv$
7: **for** each service s in $tempSrv$ **do**
8: $s.curQoS \leftarrow (c.optQoS + s.QoS)$
9: **end for**
10: **end for**
11: $addedConcept \leftarrow \varphi$
12: **for** each service s in $addedService$ **do**
13: **if** $(reverse(\mathrm{inTreeP}, \mathrm{inTreeL}, n, s, 0, -1) \setminus conceptPool) \neq \varphi$ **then**
14: $addedService \leftarrow addedService \setminus s$
15: $s.curQoS \leftarrow \infty$
16: **end if**
17: **end for**
18: **for** each service s in $addedService$ **do**
19: $addedConcept \leftarrow addedConcept \cup reverse(\mathrm{outTreeP}, \mathrm{outTreeL}, n, s, 0, -1)$
20: **for** each concept c in $addedConcept$ **do**
21: **if** $c.optQoS < s.curQoS$ **then**
22: $c.optQoS \leftarrow s.curQoS$
23: $c.optSrv \leftarrow s$
24: **end if**
25: **end for**
26: **end for**
27: $conceptPool \leftarrow conceptPool \cup addedConcept$
28: $addedService \leftarrow \varphi$
29: **end while**
30: **if** $goal \subseteq conceptPool$ **then**
31: **return** true
32: **else**
33: **return** false
34: **end if**

Theorem 2. *Consider a service composition problem with n concepts, m services, e_{in} edges from concepts to services and e_{out} edges from services to concepts, the time spent in the search algorithm is polynomial in n, m, e_{in} and e_{out}.*

Proof: The navigation time for finding direct and reverse edges of a given node in the inTree (resp. outTree) is $O(\sqrt{e_{\mathrm{in}}})$ (resp. $O(\sqrt{e_{\mathrm{out}}})$) [14]. Support in round i, the number of newly added concept is C_{add}, and newly added service is S_{add}, the time spent in round i is $C_{\mathrm{add}}\sqrt{e_{\mathrm{in}}} + S_{\mathrm{add}}(\sqrt{e_{\mathrm{in}}} + \sqrt{e_{\mathrm{out}}})$. As the total number of applied concepts and services are the total number of applied concepts and services are fixed and no new concept or service will be created in the algorithm, the algorithm will end when the goal is reached or there are no more concepts to be added. Thus, the time spent is polynomial

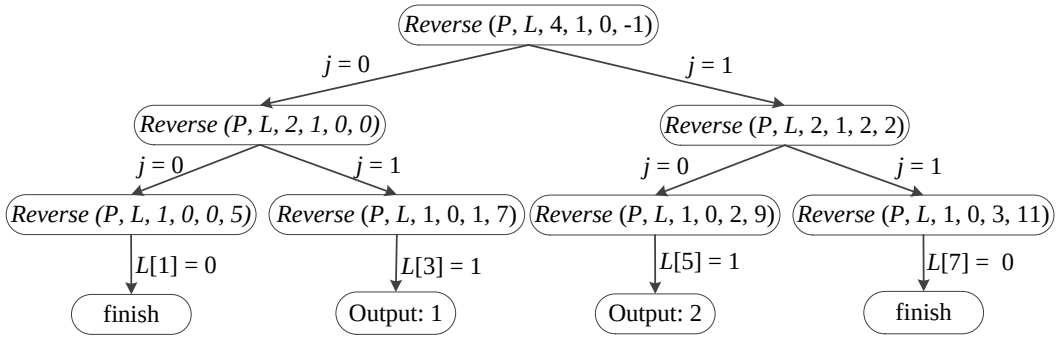

Fig. 5. Recursion trace of finding input parameters of w_2 in Figure 1

in the number of concepts n, the number of service m, e_{in} and e_{out}. ■

Algorithm 7 ***Backtracking*** performs a backtracking from the goal to the initial layer to find a suitable solution for the composition problem. The algorithm stops when the goal is a subset of the initial states. We add services which provide optimal QoS values for the goal as solution services, then we add the input parameters of that service to $goal$.

Algorithm 7 *Backtracking*

Input: $initial, goal$
Output: $solution$: a set of solution services;
1: **while** $(goal \setminus initial) \neq \varphi$ **do**
2: **for** each concept c in the $goal$ **do**
3: $cover \leftarrow cover \cup c.optSrv$
4: **end for**
5: **for** each service s in $cover$ **do**
6: $goal \leftarrow goal \cup reverse(\text{inTreeP}, \text{inTreeL}, n, s, 0, -1)$
7: $goal \leftarrow goal \setminus reverse(\text{outTreeP}, \text{outTreeL}, n, s, 0, -1)$
8: **end for**
9: $solution \leftarrow solution \cup cover$
10: $cover \leftarrow \varphi$
11: **end while**
12: **return** $solution$

IV. Experimental Evaluation

We implement the algorithms in Java and run experiments on a PC platform running 64-bit Windows 7 Operating System with an Intel Core i5 at 2.67 GHz, and 8 GB RAM. We use service challenge data set [9] to evaluate our work. For each testset, we search for solutions with either the minimum response time or maximum throughput. To make it fair, we test 10 times and get the average execution time.

The experimental results are shown in Table III. The service (resp. concept) row represents the number of services (resp. concepts) in the Testset. #service row represents the number of services in the returned solution. We use checkmarks (✓) and hyphens (-) to show whether the QoS value is correctly calculated or not. The rows P and L show the percentage of $ParentList$ and $LeafList$ in the compressed data structure. The build-time is the time spent to build the tree and search graph, the search-time represents the time spent to retrieve a solution. Compression rate, which is calculated by Equation 3, is the ratio between the number of cells after compression and the original number of cells used in the adjacency matrix. We study the proposed approach with $k = 2$ and $k = 4$.

$$compression\ rate = \frac{compact\ structure}{services \times concepts} \tag{3}$$

We observe that, this compact representation method has a good compression ratio, which helps reduce the storage requirement. From Algorithm 4 and Algorithm 5, we can see the search time is related with the height of the tree. When k is greater, the corresponding tree is shorter and wider, and as a result, the search time is shorter. However, the compression ratio is worse when k is greater, because more 0s are stored in $ParentList$ and $LeafList$ and the percentage of $LeafList$ in compressed representation increases.

TABLE III
Experimental results.

		Testset1	Testset2	Testset3	Testset4
	service	500	4000	8000	8000
	concept	5000	40000	60000	60000
	#service	8	25	10	45
	R[1]	✓	✓	✓	✓
	TP[2]	✓	✓	✓	✓
k=2	***P***	65.5%	72.9%	74.4%	74.2%
	L	34.5%	27.1%	25.6%	25.8%
	CR[3]	20%	4%	3%	3.5%
	build (s)	3	199	2943	5262
	search (s)	0.1	13	41	244
k=4	***P***	30.3%	45.2%	47.9%	47.4%
	L	69.7%	54.8%	52.1%	52.6%
	CR	32.7%	7.4%	5.9%	6%
	build (s)	1.9	80	567	1059
	search (s)	0.05	5	7.2	39

[1] R: response time (ms) as a QoS metric
[2] TP: throughput (invocations per minute) as a QoS metric
[3] CR: compression rate, calculated according to Equation 3

Figure 6 shows the comparison of search time with different values of k. We observe that with the increase of services, the search time increases slower when $k = 4$ than that of $k = 2$.

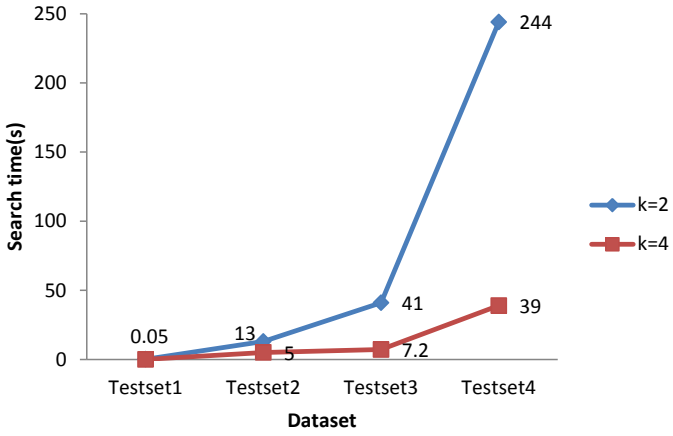

Fig. 6. Comparison of search time.

V. Related work

In this section, we survey related work from the following categories.

A. Cloud Service Composition

Though service composition has been extensively studied, cloud service composition is still a new topic and many challenging issues like real-time requirements, QoS model for cloud, network protocols are waiting to be addressed [15], [16], [17]. Duan analyzes achievable performance of cloud service provisioning and proposes a cloud service provisioning system [15]. In his system, SOA (Service Oriented Architecture) is applied in network virtualization, thus integrate networking and cloud computing system into a composite service provisioning system. Bao and Dou hold the point that, services in cloud environment are not segregate and irrelevant. Based on this consideration, they adopt a Finite State Machine (FSM) to declare the order among services in the cloud. They also propose an improved Tree-pruning-based algorithm to solve the service composition problem by creating a search tree [16]. Huang *et al.,* address the QoS-aware composition across network and cloud services problem [17]. They formulate the composition problem as a Multi-Constrained Optimal Path (MCOP) problem. Theoretical analysis is given to show the efficiency of their algorithm.

More research in this area can be found in a survey by Kourtesis *et al.* [18].

B. Optimization methods

Optimization algorithms are considered in web service composition problem to meet users' constraints and specific preferences. They can also help remove services that cannot be invoked or combine equivalent services to reduce the search space.

1) Semantic matching: To better understand users' requirements and improve correctness of returned solutions, researchers consider semantic in composition methods. Semantic interpretation helps effective service discovery and data integration on the web. Ontology is a hierarchy subdivided according to the similarities and differences among different entities. Ontology is an essence to enable semantic interpretation. Web Ontology Language (OWL) is one of the most important languages for specifying ontologies [19]. Semantic matching is widely used in web service composition process [20], [21], [22], [11]. Ontologies are used to eliminate semantic ambiguity on concepts in the AI planning process [20]. Shin *et al.* [14] define functional semantics of services, experimental results prove precision of applying functional semantics in web services. Gal *et al*. define an "integration effort" measure method to evaluate alternative solutions in the design time or search process [23].

2) QoS: In addition to fit functional requirements, to meet constraints in the composition problem and users specific preference, recent research takes consider of non-functional requirements: Quality of Service (QoS), such as response time, throughput, reputation and price. Jiang observes that QoS criteria can help prune the search space in the service composition problem. Based on this observation, a QSynth tool is proposed and implemented. Services fail to provide optimal QoS values or with worse QoS values are pruned in the forward search stage; a backward search stage is executed to generate the solution path [10]. Qiqing *et al.,* consider finding a minimum cost solution path by applying a probabilistic ant colony optimization Algorithm [24]. Yan and Chen combine Dijkstra's algorithm with a planning algorithm to solve the composition problem [2]. An attractive metric of their algorithm is: it is an anytime algorithm that may get better solutions if keeps running for longer time. Wu *et al.* apply basic form of hidden Markov model (HMM) to assess the QoS-satisfied capability in the service composition problem [25]. In our previous work [26], we propose a relational-database approach to solve service composition problem. Possible service combinations are generated and stored in a relation database. When a user request comes, SQL queries are used to obtain the solution. A local search strategy based on Artificial Bee Colony is proposed in [27]. With this method, the composition problem is transformed into a continuous space problem. The Von Neumann neighborhood topology is applied in their method to improve the quality of local search.

3) Clustering: In real world, web services may be produced by same or similar purpose, it is unavoidable that there are some overlaps of the outputs between two services. The cost increases if a lot of services with same outputs are added in the solution. This problem has motivated researchers to combine similar services in clusters [28], [29], [30]. Alrifai *et al.* use a utility function to evaluate quality of services and select skyline services to reduce the number of candidate services. Wagner *et al.* utilize a data structure to collect similar services in clusters, only root nodes of clusters are considered being added in the solution [29]. However, this classification method does not tell us what if parts of services outputs have overlap which is very common in real world web services. Rodriguez-

Mier *et al.* [30] regroup services in the same layer. For services in the same group, if all the outputs of one service are contained in the outputs of another service, the former service is deleted. Also, this classification method is too naive. The work proposed by Cai *et al.* combines clustering and correlation mining method to solve the composition problem [31]. An attractive property of this method is, a single instance of this approach may serve multiple users.

VI. Conclusion

We propose a compact web service composition system which allows algorithms to run on large data sets with lesser space requirements. QoS criteria are considered to satisfy user's constraints and preferences. To the best of our knowledge, this is the first time that compressed graph representation is applied in solving web service composition problem. In this system, edges between services and concepts are compressed and represented as two K^2-trees, one for services and input concepts and the other for services and output concepts. We search the K^2-trees for matching services and concepts. Decompression is unnecessary in forward or reverse navigation in the K^2-trees. In future work, we will explore the potential of the proposed data structure for the space and search time tradeoffs.

References

[1] J. Hoffmann, P. Bertoli, and M. Pistore, "Web service composition as planning, revisited: in between background theories and initial state uncertainty," in *AAAI'07 Proceedings of the 22nd national conference on Artificial intelligence*, vol. 2, 2007, pp. 1013–1018.

[2] Y. Yan, M. Chen, and Y. Yang, "Anytime qos optimization over the plangraph for web service composition," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. ACM, 2012, pp. 1968–1975.

[3] M. Kuzu and N. Cicekli, "Dynamic planning approach to automated web service composition," *Applied Intelligence*, vol. 36, no. 1, pp. 1–28, 2012.

[4] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, no. 1-2, pp. 281–300, 1997.

[5] N. Deo and B. Litow, "A structural approach to graph compression," in *In MFCS Workshop on Communications*, 1998, pp. 91–101.

[6] S. Grabowski and W. Bieniecki, "Merging adjacency lists for efficient web graph compression," in *Man-Machine Interactions 2*, ser. Advances in Intelligent and Soft Computing. Springer Berlin Heidelberg, 2011, vol. 103, pp. 385–392.

[7] N. Brisaboa, S. Ladra, and G. Navarro, "k^2-trees for compact web graph representation," in *String Processing and Information Retrieval*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5721, pp. 18–30.

[8] Web ontology language for web services. [Online]. Available: http://www.w3.org/submission/owl-s/

[9] S. Bleul, T. Weise, and K. Geihs, "The web service challenge - a review on semantic web service composition," *Electronic Communications of the EASST*, vol. 17, 2008.

[10] W. Jiang, C. Zhang, Z. Huang, M. Chen, S. Hu, and Z. Liu, "Qsynth: A tool for qos-aware automatic service composition," in *Web Services (ICWS), 2010 IEEE International Conference on*, July 2010, pp. 42–49.

[11] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "A dynamic qos-aware semantic web service composition algorithm," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, C. Liu, H. Ludwig, F. Toumani, and Q. Yu, Eds. Springer Berlin Heidelberg, 2012, vol. 7636, pp. 623–630.

[12] Y. Yan and M. Chen, "Anytime qos-aware service composition over the plangraph," *Service Oriented Computing and Applications*, June 2013.

[13] (2014) K-ary tree. [Online]. Available: http://en.wikipedia.org/wiki/K-ary_tree

[14] D.-H. Shin, K.-H. Lee, and T. Suda, "Automated generation of composite web services based on functional semantics," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 4, pp. 332–343, 2009.

[15] Q. Duan, "Modeling and performance analysis on network virtualization for composite network-cloud service provisioning," in *Proceedings of the 2011 IEEE World Congress on Services*, ser. SERVICES '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 548–555.

[16] H. Bao and W. Dou, "A qos-aware service selection method for cloud service composition," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 2254–2261.

[17] J. Huang, G. Liu, Q. Duan, and Y. Yan, "Qos-aware service composition for converged network-cloud service provisioning," in *Services Computing (SCC), 2014 IEEE International Conference on*, June 2014, pp. 67–74.

[18] D. Kourtesis, J. M. Alvarez-Rodríguez, and I. Paraskakis, "Semantic-based qos management in cloud systems: Current status and future challenges," *Future Generation Computer Systems*, vol. 32, no. 0, pp. 307–323, 2014.

[19] W3C OWL Working Group. (2012) OWL 2 Web Ontology Language Document Overview (Second Edition). [Online]. Available: http://www.w3.org/TR/owl2-overview/

[20] R. Akkiraju, B. Srivastava, A. Ivan, R. Goodwin, and T. Syeda-Mahmood, "Semantic matching to achieve web service discovery and composition," in *E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on*, June 2006, pp. 70–70.

[21] Y. Yan, B. Xu, Z. Gu, and S. Luo, "A qos-driven approach for semantic service composition," in *Commerce and Enterprise Computing, 2009. CEC '09. IEEE Conference on*, July 2009, pp. 523–526.

[22] B. Xu, S. Luo, Y. Yan, and K. Sun, "Towards efficiency of qos-driven semantic web service composition for large-scale service-oriented systems," *Service Oriented Computing and Applications*, vol. 6, no. 1, pp. 1–13, 2012.

[23] T. Sagi, A. Gal, and M. Weidlich, "Measuring expected integration effort in service composition," in *Services Computing (SCC), 2014 IEEE International Conference on*, June 2014, pp. 645–652.

[24] F. Qiqing, P. Xiaoming, L. Qinghua, and H. Yahui, "A global qos optimizing web services selection algorithm based on moaco for dynamic web service composition," in *Information Technology and Applications, 2009. IFITA '09. International Forum on*, vol. 1, May 2009, pp. 37–42.

[25] Q. Wu, M. Zhang, R. Zheng, Y. Lou, and W. Wei, "A qos-satisfied prediction model for cloud-service composition based on a hidden markov model," *Mathematical Problems in Engineering*, vol. 2013, 2013.

[26] J. Li, Y. Yan, and D. Lemire, "Full solution indexing using database for qos-aware web service composition," in *Services Computing (SCC), 2014 IEEE 11th International Conference on*, June 2014, pp. 99–106.

[27] X. Min, X. Xu, and Z. Wang, "Combining von Neumann neighborhood topology with approximate-mapping local search for ABC-based service composition," in *Services Computing (SCC), 2014 IEEE International Conference on*, June 2014, pp. 187–194.

[28] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for qos-based web service composition," in *Proceedings of the 19th International Conference on World Wide Web*. ACM, 2010, pp. 11–20.

[29] F. Wagner, F. Ishikawa, and S. Honiden, "Qos-aware automatic service composition by applying functional clustering," in *Web Services (ICWS), 2011 IEEE International Conference on*, July 2011, pp. 89–96.

[30] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "Automatic web service composition with a heuristic-based search algorithm," in *Web Services (ICWS), 2011 IEEE International Conference on*, July 2011, pp. 81–88.

[31] H. Cai, L. zhen Cui, Y. Shi, L. Kong, and Z. Yan, "Multi-tenant service composition based on granularity computing," in *Services Computing (SCC), 2014 IEEE International Conference on*, June 2014, pp. 669–676.